LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-214107

Extracting Critical Path Graphs from MPI Applications

M. Schulz

July 28, 2005

IEEE Cluster 2005
Boston, MA, United States
September 27, 2005 through September 30, 2005

Extracting Critical Path Graphs from MPI Applications

Martin Schulz
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
schulzm@llnl.gov

Abstract

The critical path is one of the fundamental runtime characteristics of a parallel program. It identifies the longest execution sequence without wait delays. In other words, the critical path is the global execution path that inflicts wait operations on other nodes without itself being stalled. Hence, it dictates the overall runtime and knowing it is important to understand an application's runtime and message behavior and to target optimizations.

We have developed a toolset that identifies the critical path of MPI applications, extracts it, and then produces a graphical representation of the corresponding program execution graph to visualize it. To implement this, we intercept all MPI library calls, use the information to build the relevant subset of the execution graph, and then extract the critical path from there. We have applied our technique to several scientific benchmarks and successfully produced critical path diagrams for applications running on up to 128 processors.

1 Motivation

The critical path of a parallel application is a fundamental metric for its execution. It represents the longest execution sequence without wait delays and hence dictates the complete overall runtime: adding to the critical path will directly increase the runtime, while shortening or optimizing sections of the critical path will result in a reduction of the overall runtime. On the other hand, changing the execution time of program sections not on the critical path will not affect the program runtime.

Knowing the critical path allows the user to characterize the application and its communication behavior, explore differences in the execution behavior with respect to scaling and parameter changes, and identify potential bottlenecks or performance problems. It is also a prerequisite for dealing with scalability problems in performance analysis and optimization. The increasing number of nodes in modern clusters, often reaching thousands, makes it impossible to investigate, manually analyze, or often just visualize the performance data collected from all nodes. Knowing the critical path, however, enables us to restrict optimization efforts to the relevant parts of the execution.

In this paper we present an hybrid online/post-mortem approach to identify, isolate, and visualize the critical path of MPI applications. During the online component we gather information on each processor locally, in order not to add additional communication requirements. After program termination we combine the individual local results for global analysis.

In order to implement our approach, we abstract the program as an *execution graph* in which all MPI communication operations are shown as nodes. Directed edges represent both sequential execution within one processor as well as message events between processors. The critical path is then a subset of the complete execution graph. To extract it, we build relevant subsets dynamically at runtime using an MPI wrapper library that traces all MPI communication operations. At each receive operation, we identify the execution graph edges that do not incur a wait time and are hence potentially part of the critical path; we store them as part of a local subgraph. After the termination of the application, we merge these subgraphs, identify which of the potential edges stored in the local graphs are actually part of the completed global critical path, prune the remaining nodes, and export to result in a portable graph format for visualization and further processing.

This graph shows all sections in the program's execution that contribute to the critical path as well as all communication events that cause the critical path to transit between processors. The graph form has the further advantage, in contrast to profiling data or tabular representations, that it retains much of the structural information of the initial application execution and with that eases further high-level analysis steps.

[0]This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-CONF-214107).

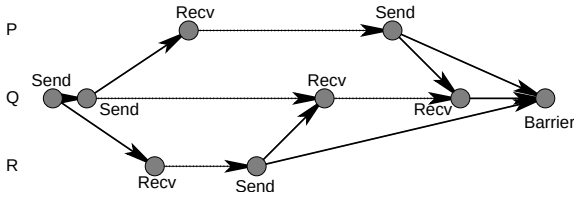


Figure 1. Sample Execution Graph.

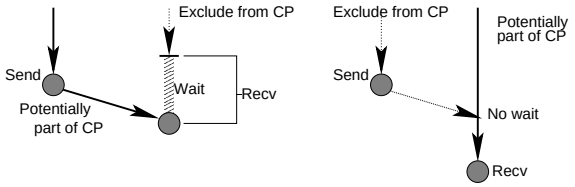


Figure 2. Wait scenarios at receive operations.

2 Critical Paths in MPI Applications

To analyze the critical path of an MPI application, we represent an instance of the application execution as a directed graph, which we will refer to as the *execution graph*[1]. This graph contains a node for each MPI operation (e.g., *MPI_Send* or *MPI_Recv*) and edges between send and receive pairs as well as between nodes that are in consecutive execution order. We will refer to the former as *communication edges* and to the latter as *program edges*. In addition, we will add a global node for each barrier with communication edges to and from each processor, and also introduce such a barrier node at the end of the program to capture program termination. Figure 1 shows an example of such a graph. In this example, we see a program execution on three processors P, Q, and R. Dotted arrows show program edges, while solid arrows show communication edges.

The critical path of a program is the path through the execution graph that determines the program's overall runtime. This path has the property that any delay on this path will directly translate into a longer runtime, while any optimization along this path will result in an overall reduction in runtime. The traditional approach to detect and extract the critical path is to annotate all program edges with the CPU time (excluding wait operations) that was measured during the program execution along these edges. The critical path is then computed as the longest path from the start node to the end node with respect to these annotations. This requires a global search across the whole graph, which is often time and resource consuming.

The critical path, however, can also be piecewise identified by examining all receive operations using the following two observations:

> *The critical path can not contain program edges leading to receive nodes that incur a blocking wait time.*

This is true, because any delay along this program edge (up to the wait time incurred at the receive) will only decrease the wait time, but not delay the overall execution. This is contrary to the definition of the critical path.

[1]Similar representations are sometimes also called Program Activity Graph (PAG), Space-Time Diagram, or Event Diagram.

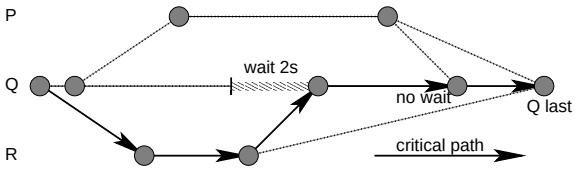


Figure 3. Finding the critical path.

Similarly, the following statement holds as well:

> *The critical path can not contain communication edges, which do not lead to blocking wait times.*

In this case, the message had already arrived at the time the receive operation was called and hence was immediately received. Delaying the message (up to the time difference between the actual message arrival and the matching receive operation) can again be tolerated without runtime increase.

These two scenarios are shown in more detail in Figure 2. Using these rules, we can extract the critical path from the message graph by excluding edges not on the critical path and recording only the remaining ones as a *partial execution graph* without requiring a global search. When we combine these edges using backtracking from the termination node, we are then able to gather all components of the critical path and create a single representation of the complete critical path graph.

Figure 3 illustrates this further using the execution graph from Figure 1: process Q waits at its first receive operation for a message from process R. Hence, any delay in process Q (up to the wait time) can be compensated by waiting less time, while any delay in process R will cause Q to wait longer. Hence we store the edges on R as potentially being on the critical path. During the second receive, however, no wait occurs, i.e., the message has arrived before the receive operation is executed. Consequently process P is not part of the critical path and only the program edge of Q is stored. Similarly we classify the incoming edges at barrier nodes: the last process to arrive does not have to wait and hence is added to the graph.

3 Implementation

The critical path is a global property and hence some form of global operation is required to extract it. In this section we describe our approach to identify the critical path without having to introduce additional communication paths or requiring a global search, and with no code changes to the target application.

3.1 Approach

We construct the critical path in two steps: first we collect local information at runtime on each processor and create subgraphs with critical path information. Second, we use offline, post-mortem analysis to merge the local subgraphs into a global graph and extract the critical path.

The architecture of our system is illustrated in Figure 4. To extract the critical path from a running application we must intercept all communication events. For this, we interpose our implementation into any MPI communication call using a wrapper library. Within the wrapper we apply the rules discussed in Section 2 at each receiving node. Based on this information we build local, partial execution graphs for each processor containing all edges that could potentially be part of the global, critical path.

Once the program is complete, each processor writes its partial execution graph to storage. In a postprocessing step, our tool reads all local graphs, merges them, performs several automatic graph analysis and reduction steps, and then exports the final graph in the portable and open graph format GML (Graph Modeling Language [7]).

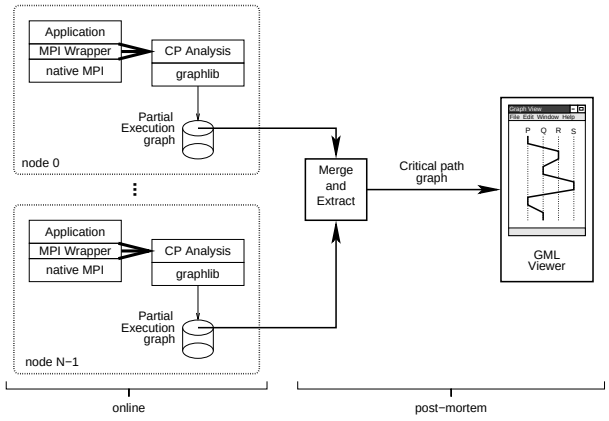


Figure 4. Architecture of the toolset to extract the critical path.

3.2 Graphlib

To enable efficient, yet high-level graph handling, we have developed *graphlib*, a library to create and manipulate multiple independent, arbitrary graphs. Besides the mandatory graph manipulation routines to create and delete graphs and to add and remove nodes and edges, *graphlib* contains several high-level analysis routines, as well as the ability to export graphs into several portable, open graph formats, including DOT [11] and GML [7].

Within *graphlib* all information about nodes and edges is stored in linked lists of fragments. Each fragment can hold a static number of entries and a graph can have an arbitrary number of node and edge fragments. This provides a suitable trade-off between memory management overhead and memory fragmentation.

In order to support fast node or edge addition and removal, while maintaining memory efficiency, each graph in *graphlib* also maintains free lists for both nodes and edges, which directly point to the individual entries inside the fragments. As a consequence, both node and edge deletions and additions can be performed in constant time.

3.3 Send/Recv Operations

Using *graphlib*, we maintain a local, partial execution graph on each processor of the MPI application. Starting with an empty graph at program start, we add a node at each send and receive operation. In order to establish a global, partial order, we associate each node with a local timestamp and include this timestamps in any send operation in the form of piggybacked data[2]. At each receive operation, the local timestamp is compared to the timestamp received as part of the incoming message and set to the maximum of the two values.

At each receive operation, we apply the rules introduced in Section 2. For this, we need to query whether a message has arrived prior to posting a matching receive or not, in order to decide which edge is potentially part of the critical path. Unfortunately, MPI does not provide a direct mechanism to determine this. Hence, we have to rely on indirect observations to distinguish between these two cases: we tried both deducing this from the measured execution time of a blocking MPI operation and querying the MPI layer prior to posting the receive using a probe operation. Both mechanisms work, but we found the latter mechanism to be more reliable.

It should be noted, though, that a misclassification, which can happen with both mechanisms if wait times are

[2]We use the piggyback mechanism mentioned in the C^3 system [15]. It is based on constructing new datatypes which augment the sendtype with the piggyback data.

short, is not critical. In this case the execution time difference between the two potential paths is small — large differences stemming from large wait times are correctly detected in both cases.

Once this classification is made, we can add the respective edge to the local graph: in the case the receive is executing a blocking wait, i.e., the receive is posted before the message arrives, we add the incoming communication edge; in the other case, i.e., the message is already present before reaching the receive, we add the program edge to the previous node on the same process.

At each send operation, we add a program edge to the previous node on the same process if and only if that previous node is part of the existing local subgraph, i.e., is potentially on the critical path. Asynchronous sends are handled in the same way as blocking sends, while asynchronous receives are ignored and the analysis for receives is instead applied to the corresponding wait operations.

Collective communication operations are treated as a set of individual point-to-point messages, since this provides us with a clear indication of which node inflicts a wait time on the overall process. Barriers, on the other hand, are treated separately: in this case we measure the execution time of the barrier on all nodes and add a communication edge from the node with the lowest barrier time, since presumably this process reached the barrier last and is hence guaranteed to be part of the critical path. For this, however, we have to add an *MPI_Allreduce* for each barrier to find this minimal waittime.

In addition to the actual node information, we also store information about the call stack of the MPI function associated with each node by unwinding the local stack. In the completed graph, the user can then use this information to identify the nodes on the critical path and with that the parts of the code with the highest potential for optimizations.

3.4 Graph Analysis

After the termination of the application, each processor writes its partial graph to disk. A postprocessing tool based on *graphlib* reads all partial graphs, merges them into a single one, analyzes the graph to extract the critical path, and exports it into GML.

The merge utility applies a set of analysis and graph reduction steps. We illustrate this in Figure 5 using a small sample application. All of these graphs are generated by *graphlib*'s GML export filter and visualized using the freely available graph editor *yed* [18]. In all graphs, each column of nodes represents one process and the virtual time (i.e., the timestamps) grows monotonically from top to bottom. Graph a) shows the unoptimized graph, the following graphs the results of the consequent optimization steps described below:

Critical Path Detection: To detect the critical path we start at the final barrier node recorded right before program termination and backtrack on the edges identified as potentially being on the critical path. The critical path is complete once the initial node in any processor is reached (as shown in graph b). It is guaranteed that exactly one such path exists since any receive operation has exactly one potential critical edge associated with it. Hence, in each step of the backtracking process it is guaranteed that exactly one potentially critical edge can be found as incoming edge to the node in question. During this process we color all nodes and edges on the critical path red, leaving all nodes and edges in the remaining graph gray and green respectively.

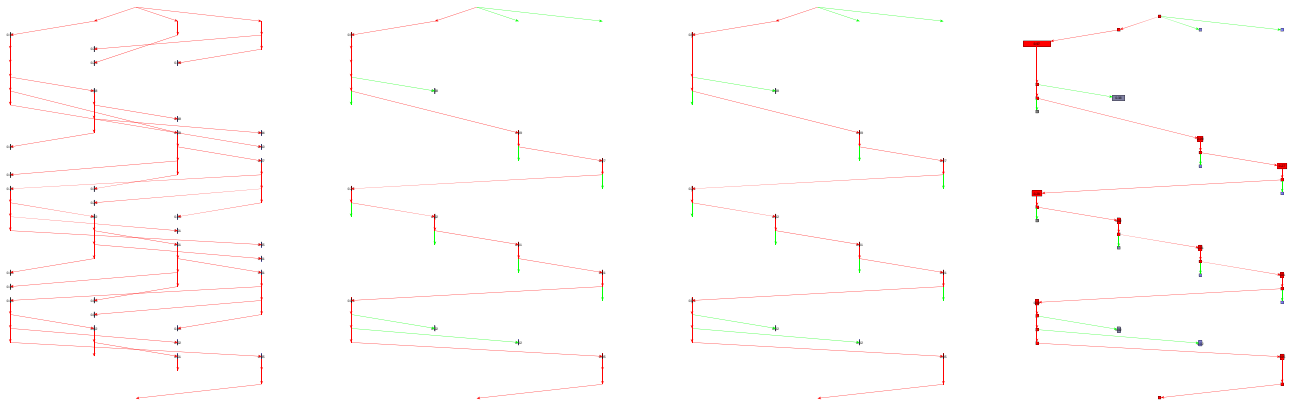
Graph Pruning: Sections of the graph that are not part of the critical path can be removed by pruning. However, edges leading away from the critical path are potentially relevant from a program optimization point of view, since the wait times that are associated with them indicates the delay the critical path inflicts at this point on the program execution and hence are directly correlated with the optimization potential. We therefore keep these nodes with a distance of one from the critical path.

To prune the graph, we follow the critical path and apply a pruning step to each node pointed to by an edge from a node on the critical path. In this step, we delete all subgraphs from that node, leaving only the node itself and the edge leading to it. For efficiency reasons, this pass is combined with the actual critical path detection and coloring. The result of this pruning step is shown as graph b).

Graph Collapsing: Pruned graphs often contain large node chains connected by program edges, i.e., nodes representing consecutive MPI operations within one process. This is especially true for codes with large number of processors (e.g., caused by a set of sequential send operations to all processors). However, it is only relevant to record at which points the critical path transitions between processors. Hence, these chains of program edges can be collapsed and each replaced by a single edge.

To implement this, we scan all nodes of the graph for those that are only connected to a single incoming and a single outgoing program edge. These nodes, as well as the outgoing edges, are deleted and their incoming edge is extended to the node originally following the outgoing edge. Graph c) is the resulting collapsed graph.

Node scaling: During the graph generation we record the wait time for each receive operation at that node. In this last graph manipulation step, we first compute the maximum and minimum value of these wait times and then use these results to scale the node parameters. In the final graph, these parameters are then used to render the size of the



a) After merge b) After Pruning c) After Collapse d) After Scaling

Figure 5. Steps in the Graph Analysis (graphs generated with *yed* [18]).

nodes: large nodes represent long wait times, small ones short waits. The final graph is shown as graph d).

The graph pruning and collapsing steps are critical to reduce the size of the final graph. When executed solely as a postprocessing step, we have to create and store the complete local execution graphs at runtime. Further, the postprocessing step has to merge all subgraphs, which leads to significant resource requirements. It is therefore important to prune and collapse the graph as much as possible already at runtime.

However, since the critical path can only be determined using backtracking from the final node in the graph, full pruning at runtime is not possible. Nevertheless, we can prune partial graphs that can be excluded because they are guaranteed not to be on the critical path. For this, we examine again the two cases of edge classification, as shown in Figure 2 and described in Section 2. In the case that the program edge leading to the receive node is excluded from the critical path, we can delete this edge and recursively the path leading to this edge until we reach a node that can again potentially be part of the critical path.

In the other case, that the communication edge is excluded from the critical path, we can also delete this edge and the path leading to it. Unlike the first case of deleting a program edge, which is a local operation, the deletion of a communication edge requires deleting part of the graph stored on another node. To avoid additional communication, we do not send the prune request right away, but rather store it in a local buffer. During the next application message to this node, we add this prune request to the piggyback data. The receive operation on any node checks for these extra requests in the receive wrapper and executes it. This operation can then in turn lead to another prune request on a different node, which again is stored in the local buffer waiting for the next message send to that particular node.

It should be noted that, in case this local buffer is full, pruning requests can be dropped without impacting correctness. Similarly, pruning requests that remain at program termination do not have to be executed. Unpruned parts of the graph will be removed during the postprocessing steps.

3.5 Limitations

Our approach currently does not work for applications using busy wait loops in cooperation with *MPI_Test* operations. Here, the analysis can not detect the reason for application progress and associate the correct edges with the critical paths. The only way solve this problem is to introduce appoproriate user annotations that identify when the program is idling and waiting for a message to arrive or when it is in a busy wait loop still executing useful work. In the former case, the edge associated with the incoming message is part of the critical path, while in the latter case it is not.

Further, long running experiments are currently not feasible with our approach, since we collect the critical path graphs on each node throughout the entire runtime and write the complete graph at the termination of the application. This means that the resource requirements to store the graph naturally grow over the runtime of the application, even with the above mentioned pruning techniques in place. For this, we are extending the approach to periodically store the local subgraphs and reset the graph generation. This reduces the pressure on resources at runtime, at the cost of increased postprocessing.

In case of applications with tight synchronization, e.g., in the form of repeated barrier operations, these synchroniza-

tion points form a natural location for these partial graph saves. At these points, we know that the graph node representing the global synchronization has to be part of the critical path and hence we can apply the extraction and pruning steps independently between each synchronization point. This, however, requires an extension of *graphlib* to enable the concatenation of multiple, partial or phase-wise critical paths.

Nevertheless, it will not be possible to fully mitigate these resource problems. Our approach targets the explicit generation of critical path graphs with the goal to study their topology. The size of this graph will by its nature always be linear to the number of MPI operations per processor. If this is a problem, the user should use to critical path profiling methods like described in [8, 9]. Those incur less overhead and resource requirements, but at the trade-off of having less expressiveness in the results.

4 Examples and Evaluation

We conducted all of the following experiments on *MCR*, a 1152 node cluster at LLNL. Each node is equipped with two 2.4 MHz Dual Xeon processors, 4 GByte of memory, and is connected using Quadrics's QsNet (Elan-3). The system uses the CHAOS–2 Linux distribution, which is based on Redhat Enterprise Linux 3. The benchmarks were compiled using gcc 3.2.3 and linked to Quadrics's MPI implementation, which is based on MPICH–1, but modified to take advantage of Quadrics's QsNet.

We present results using two well-known scientific applications: SMG2000 and HPL. The former is a Semicoarsining Multigrid Solver based on the Hypre library [5], and the latter is a portable high-performance implementation of the Linpack benchmark [13].

4.1 Stability

The critical path of an application is a runtime property and hence can vary from run to run of an application, even when used with the same application parameters. This is especially true, if the application contains message nondeterminism, e.g., in the form of wildcard receives. However, in order to be useful for static program analysis and optimization, these differences should be minimal.

Figure 6 shows the critical path of four separate SMG2000 runs with identical parameters on four processors after pruning, collapsing, and scaling. The four graphs are almost identical; only during initialization and termination small variations exist. However, the concrete wait times (shown by the different node sizes), do vary from run to run. This result shows that the critical path is a quite stable program characteristic for SMG2000, despite timing differences in consecutive runs. Similar observations can also be made for HPL.

4.2 Interpretation

The critical path of SMG2000 (Figure 6) indicates at least two distinct phases. First, the critical path shows an irregular pattern across all nodes and with longer sections in which the critical path is dominated by a single processor (rank two). This is followed by a phase where the critical path of SMG2000 only transitions between processes one, two, and three. Rank zero is, except for minor parts, not affected. Interesting is also to notice that long delays can mostly be observed in the first phase of the program (large nodes), while in the remainder of the program all wait times are small and roughly equal. This indicates that optimizations on the critical path are likely to have a larger impact in the first phase of this program.

4.3 Scalability

In this section we look at the scaling properties of critical paths. For this, we run HPL on 32, 64, and 128 processors. The resulting graphs can be seen in Figure 7. All graphs show similar patterns. Wait times are fairly uniformly distributed and no single node plays an outstanding role. Towards the end of the execution, we see a stair-case effect indicating some kind of serialization. The operations after this serial phase show a logarithmic communication pattern as it is typical in collective operations.

In general, the graphs show that the shape of the critical path of HPL does not significantly change when scaled to larger numbers of processors. This indicates that the critical path can be a useful property that holds across scaling boundaries and hence enables a prediction of code characteristics on system sizes not tested earlier.

4.4 Graph Sizes

Table 1 shows the number of nodes for both benchmarks running on various system sizes. For each configuration, the table shows the size for an unpruned, full graph, after pruning subgraphs not on the critical path, and after the final collapsing operation. It should be noted that the number of edges is always one less than the number of nodes since all graphs in questions are trees. Consequently, we are only reporting node numbers.

The data shows a significant difference in the number of nodes between the full graph, which represents a naive implementation, and the pruned graph. The latter, as well as the collapsed graph, show a significantly reduced graph size, which enables a more efficient and easier handling and visualization of the resulting graph.

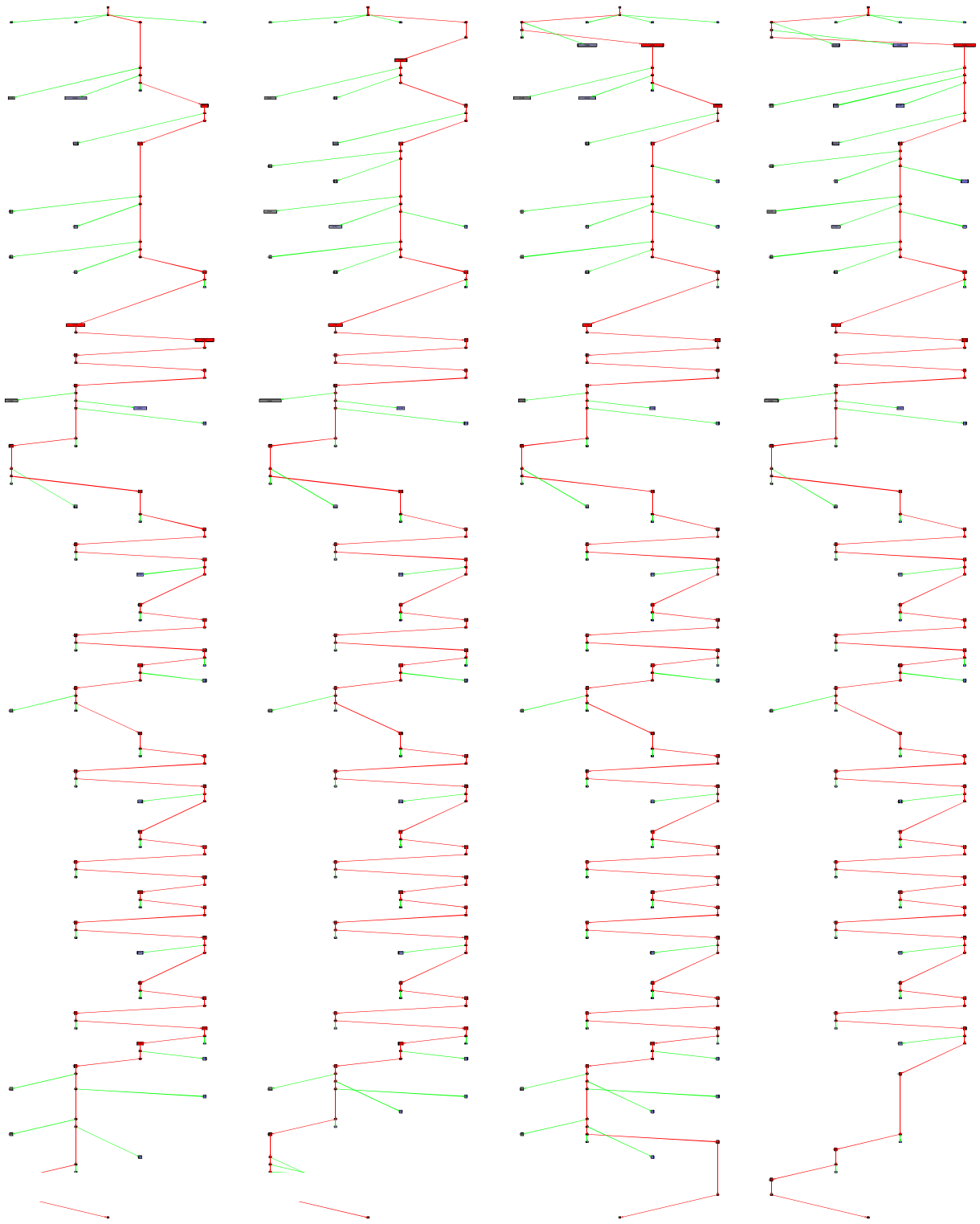


Figure 6. Critical path graphs for SMG2000 (4 processors) of repeated runs with identical parameters (graphs generated with *yed* [18]).

a) 32 processors b) 64 processors c) 128 processors

Figure 7. Critical paths of HPL on various system sizes (graphs generated with *yed* [18]).

Benchmark	Number of CPUs	Full graph	Pruned graph	Collapsed graph
SMG2000	4	495	165	109
SMG2000	8	1863	324	198
SMG2000	16	6767	595	340
SMG2000	32	25079	1044	442
SMG2000	64	95551	1745	958
SMG2000	128	371719	3392	825
HPL	4	538	219	214
HPL	8	1434	329	319
HPL	16	3546	454	446
HPL	32	8410	587	573
HPL	64	19418	752	732
HPL	128	43994	1023	1005

Table 1. Number of nodes used in the generated critical path graphs.

Besides showing the feasibility of the approach, these numbers also indicate its opportunities. The graph analysis algorithm discussed in this paper is able of reducing the full trace information in the form of the complete program execution graphs to a minimal set of nodes that contain the important information, while discarding the rest. Further analysis steps by further tools or manual examination are therefore made easier.

4.5 Overhead

Table 2 shows the overhead caused by adding our system to SMG2000. This includes the overhead caused by our transparent piggybacking, the wait-time observations at receives, as well as the actual subgraph generation. We compute these numbers by comparing the application execution time with our mechanism to the execution time of an unmodified version of SMG2000 at same optimization levels (-O2). The table shows both relative and absolute numbers for 32 and 64 processors on a range of working set sizes per processor.

For small working set sizes and short runtimes, the initialization overhead dominates resulting in large overheads of over 40%. With increasing working sets (which are typical in realistic usage scenarios of SMG2000) this overhead drastically reduces to acceptable levels of under 8%. For very large working set sizes, the overhead rises again slightly, most likely caused by the larger execution graphs, which need to be extracted and stored. However, at those large working set sizes, the overhead is still only around 12%. The same experiments on HPL resulted in similar overhead numbers of 4%-11% at large enough runtimes. In general, we expect to reduce these numbers with further optimizations within *graphlib* and by investigating more aggressive pruning.

5 Related Work

The concept of a critical path as a means to identify performance problems and to steer optimizations has been explored in several areas. Barford and Crovella [3] use it to study the performance of TCP in the context of HTTP transactions. Tullsen and Calder [16] discuss the use of critical path information to reduce dependencies in binary codes and also Intel integrates a similar functionality into their Thread Checker tool [17].

In the context of parallel, message passing programs, Alexander et al. [2] compute *Near Critical Paths* using search algorithms on execution graphs in which each program edge is weighted with the computational complexity of the corresponding program section. In contrast to our work, they require global searches across the whole graph, which is not necessary in our approach. Instead, a straightforward backtracking through the graph is sufficient.

Closely related to the work presented here is the critical path profiling by Hollingsworth [8, 9]. This work employs a sophisticated online mechanism built on top of dynamic instrumentation to collect the necessary input. The information collected in this project is, however, geared towards aggregated metrics and based on a per-function evaluation. It is not possible to extract the full structural information of the critical path and to visualize it. On the other hand, though, restricting information to such aggregates dramatically reduces the pressure on resource requirements.

Besides the generation and analysis of critical paths, several other approaches exist to determine execution characteristics of parallel programs and to find potential optimization points. The traditional approach is the generation of trace files, which are then analyzed post-mortem, mostly manually with the help of visualization tools. Examples for this can be found in Tau [4], Vampir [12], or Paraver [14]. The PARADIS project [6] uses event graphs, which are similar to the execution graphs used in this paper, to detect performance anomalies and bottlenecks. Ahn and Vetter [1] use statistical methods, e.g., cluster analysis, to group processors of similar execution behavior and Nagel et al. [10] apply statistical methods to predict upper and lower bounds for the cost of MPI routines.

6 Conclusions and Future Work

The critical path is a runtime characteristics of an execution instance of a parallel program. It identifies the longest path through the execution graph without waits and hence is the path that dictates the overall execution time. Knowing this critical path is useful to understand the program and communication behavior of an application. Further, the critical path can be used to steer optimization efforts, since any

Working set size	30^3	40^3	50^3	60^3	70^3	80^3
32 processors	43.55%	33.93%	20.53%	10.38%	6.07%	8.64%
(absolute)	1.55s	2.58s	3.19s	2.69s	2.65s	5.30s
64 processors	46.34%	37.00%	21.28%	7.88%	11.79%	12.02%
(absolute)	1.78s	2.96s	3.42s	2.15s	5.01s	7.51s

Table 2. Overhead for computing the critical path for SMG2000.

reduction of the critical path directly translates into an overall reduction in runtime.

In this paper, we presented a toolset to automatically analyze the execution of MPI applications and to detect and isolate the critical path. This is done in a two step process: first each processor generates a local subgraph, which is then merged post-mortem with all other subgraphs into a global execution graph. We then use this graph to identify the critical path and export the result into a public and portable graph format. Using existing graph viewers, the user can then inspect the critical path.

We have demonstrated this ability using two well-established scientific benchmarks (SMG2000 and HPL) on up to 128 processors. For both applications, the critical path revealed details about the communication behavior of the respective application without requiring explicit knowledge of the source code.

We will use the techniques presented here as the basis for our future work on scalable performance analysis and application characterization. For the former, we will combine the toolset represented here with existing sequential performance tools by applying these tools solely to the critical path and thereby reducing the amount of gathered data. As for the latter, we will apply machine learning techniques, as well graph analysis algorithms to identify phases or sections of critical paths and to compare critical paths for multiple executions with different parameters.

References

[1] D. Ahn and J. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of IEEE/ACM Supercomputing '02*, Nov. 2002.

[2] C. Alexander, D. Reese, and J. C. Harden. "near-critical path analysis of program activity graphs". In *MASCOTS*, pages 308–317, 1994.

[3] P. Barford and M. Crovella. Critical path analysis of TCP transactions. *ACM SIGCOMM Computer Communication Review*, 31(2):80–102, 2001.

[4] R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, Aug. 2003.

[5] R. Falgout and U. Yang. hypre: a Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, pages 632–641, Apr. 2002.

[6] C. Glasner, E. Spiegl, and J. Volkert. PARADIS: Analysis of Transaction-Based Applications in Distributed Environments. In *Proceedings of the 5th International Conference in Computational Science (ICCS), Part II, LNCS vol. 3515*, pages 124–131, May 2005.

[7] M. Himsolt. "gml: A portable graph file format". Technical report, Universität Passau, Germany, 1995.

[8] J. Hollingsworth. An Online Computation of Critical Path Profiling. In *Proceedings of the 1st ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 11–20, May 1996.

[9] J. Hollingsworth. Critical Path Profiling of Message Passing and Shared-Memory Programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1029–1040, 1999.

[10] M. Kluge, A. Knüpfer, and W. Nagel. Statistical Methods for Automatic Performance Bottleneck Detection in MPI Based Programs. In *Proceedings of the 5th International Conference in Computational Science (ICCS), Part I, LNCS vol. 3514*, pages 330–338, May 2005.

[11] E. Koutsofios and S. C. North. *Drawing graphs with dot*. Murray Hill, NJ.

[12] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[13] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack be nchmark for distributed-memory computers. Available at http://www.netlib.org/benchmark/hpl/.

[14] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31, Amsterdam, 1995. IOS Press.

[15] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *Proceedings of IEEE/ACM Supercomputing '04*, Nov. 2004.

[16] D. Tullsen and B. Calder. Computing Along the Critical Path. Technical report, UC San Diego Technical Report, 1998.

[17] I. Website. Intel (R) Thread Checker 2.1. http://www.intel.com/software/products/threading/tcwin/, 2005.

[18] yWorks. YEd — Java Graph Editor. http://www.yworks.com/en/products_yed_about.htm, May 2005.